Chapter 8
Deadlocks

Da-Wei Chang

CSIE.NCKU

Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "Operating System Concepts", 10th Edition, Wiley.

Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows
  - request
  - use
  - release
- A deadlock example
  - P1 uses R1, requests R2
  - P2 uses R2, requests R1

Deadlock Characterization

Deadlock can occur if four conditions hold simultaneously.

- **Mutual exclusion**
  - only one process at a time can use a resource
- **Hold and wait**
  - a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**
  - a resource can be released only voluntarily by the process holding it
- **Circular wait**
  - there exists a set of waiting processes $\{P_1, P_2, \ldots, P_n\}$ such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_1

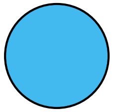
Resource-Allocation Graph

A set of vertices V and a set of edges E

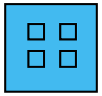
- V is partitioned into two types
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- *request* edge – directed edge $P_i \rightarrow R_j$
- *assignment* edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

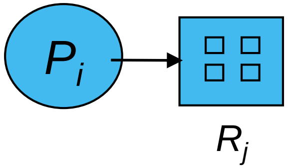
- Process



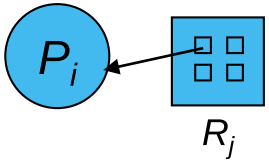
- Resource Type with 4 instances



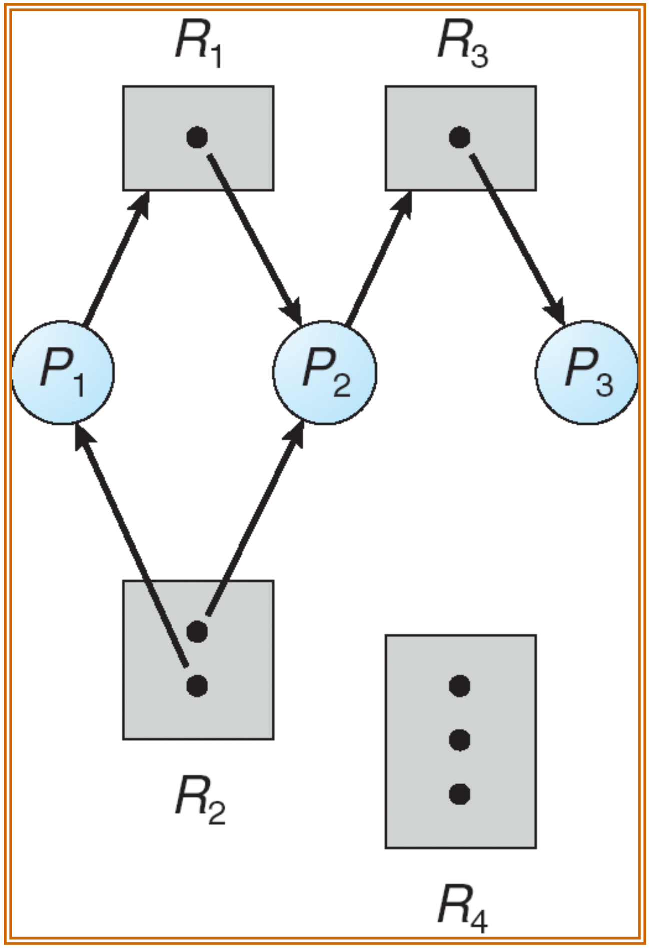
- P_i requests an instance of R_j



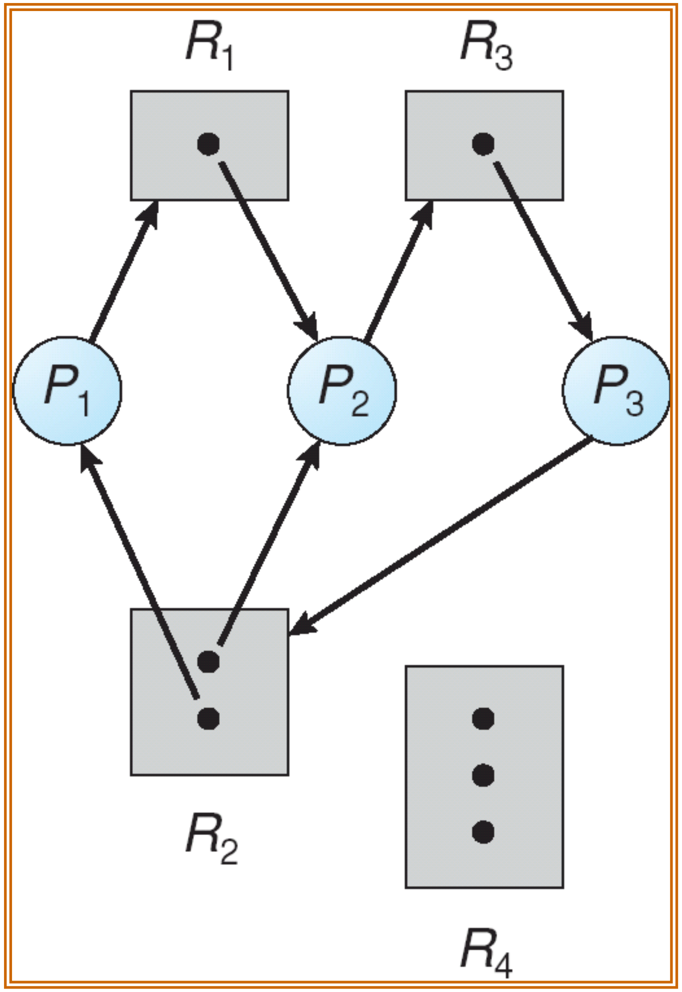
- P_i is holding an instance of R_j



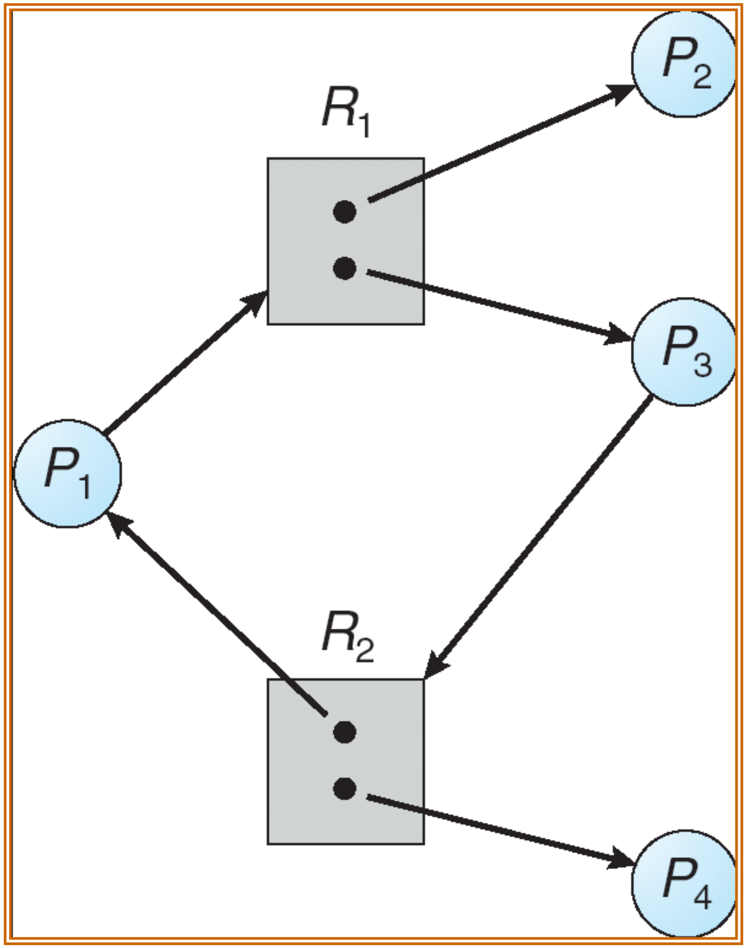
Example of a Resource Allocation Graph



Resource Allocation Graph with a Deadlock



Resource Allocation Graph with a Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
  - Deadlock prevention or avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems (UNIX, Windows..)
  - applications have to deal with deadlocks themselves

Deadlock Prevention

- **Restrain the ways request can be made**
  - Ensure at least one of the four conditions cannot hold
- **Mutual Exclusion**
  - not required for sharable resources; must hold for non-sharable resources
- **Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated **all** its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization
  - Starvation possible
    - A process that needs several popular resources may wait infinitely

Deadlock Prevention (Cont.)

- **No Preemption –**
  - If a process holds some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are released (i.e., **preempt** the resource holding)
    - Preempted resources are added to the list of resources for which the process is waiting
    - State rollback may be required
    - Process will be restarted only when it can regain its old resources, as well as the new one that it is requesting
- **Circular Wait –**
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
    - Lock-order verifier. example: witness (BSD)

Deadlock Avoidance

- Check if the admission of the current request may lead to a deadlock state (specifically, a circular-wait condition)
  - Yes ➔ do not admit the request
  - Requires that the system has some additional *a priori* information available
  - Simplest and most useful model requires
    - each process declare the *maximum number* of resources of each type that it may need
  - Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

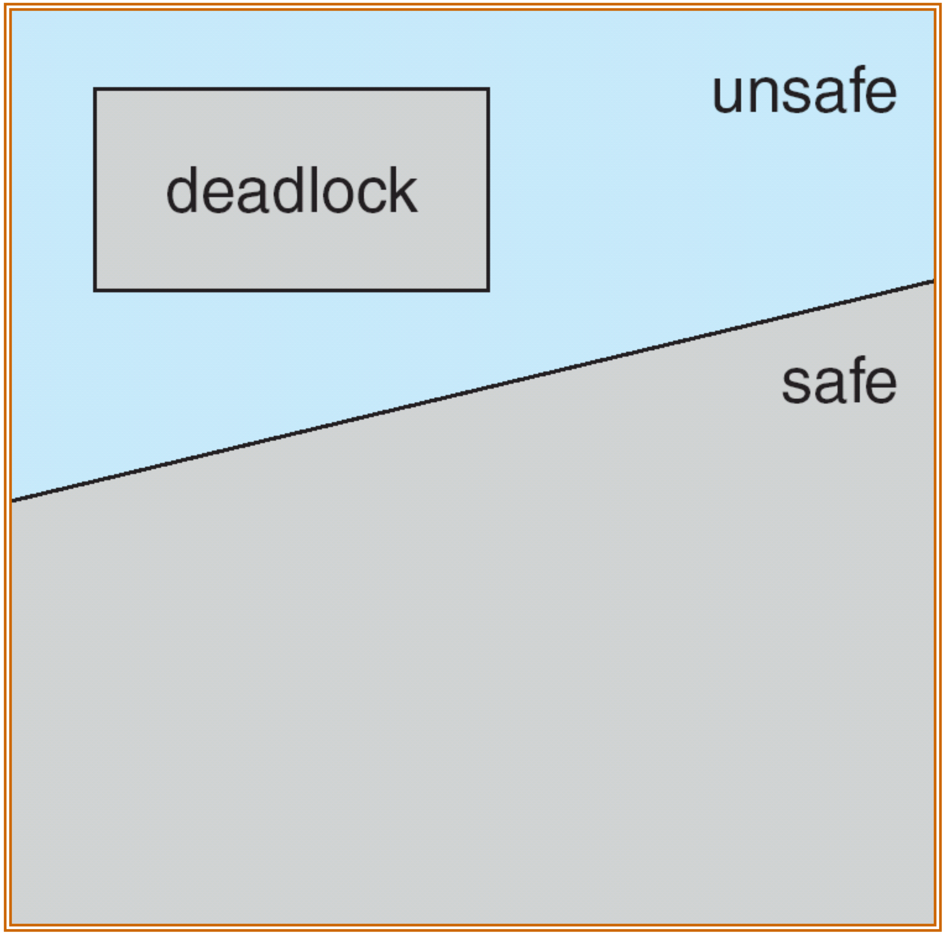
- When a process requests an available resource, system must decide if the immediate allocation leaves the system in a safe state

- System is in safe state if there exists a safe sequence of all processes
  - Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, where j<i.
    - If P_i's resource needs are not immediately available, then P_i can wait until all P_j have finished.
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state

Safe, Unsafe, and Deadlock States



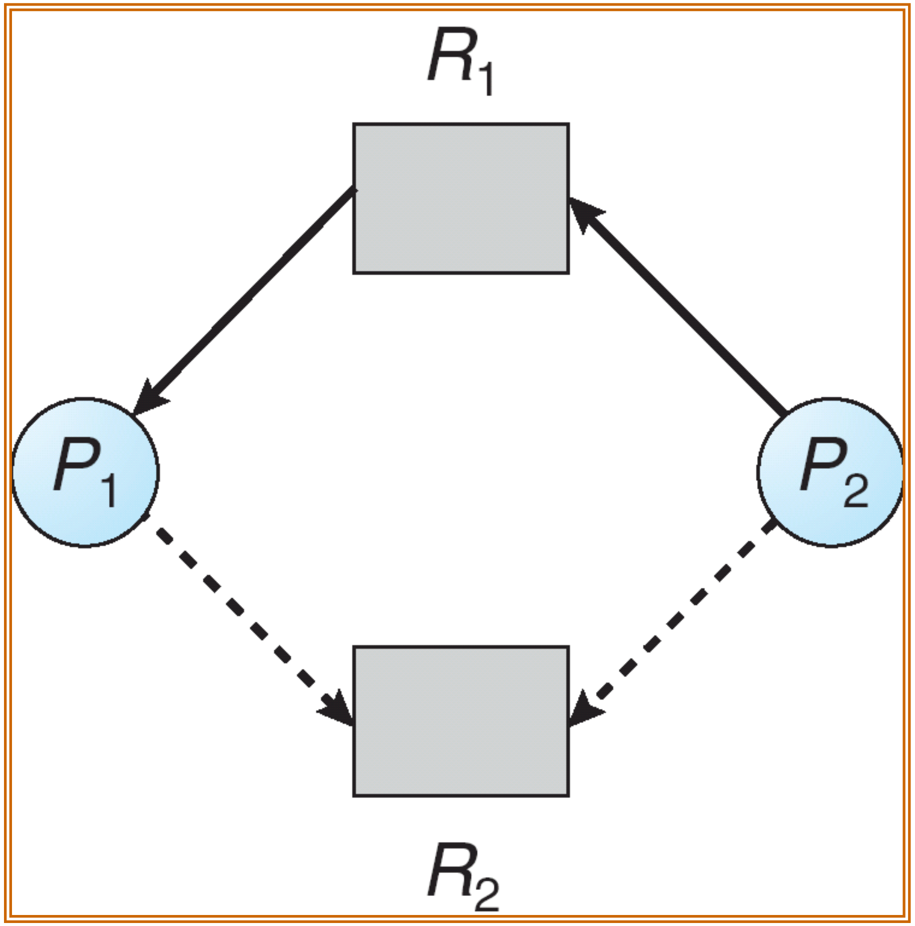
Deadlock Avoidance Algorithms

- Resource-Allocation Graph Algorithm
- Banker's Algorithm

Resource-Allocation Graph Algorithm

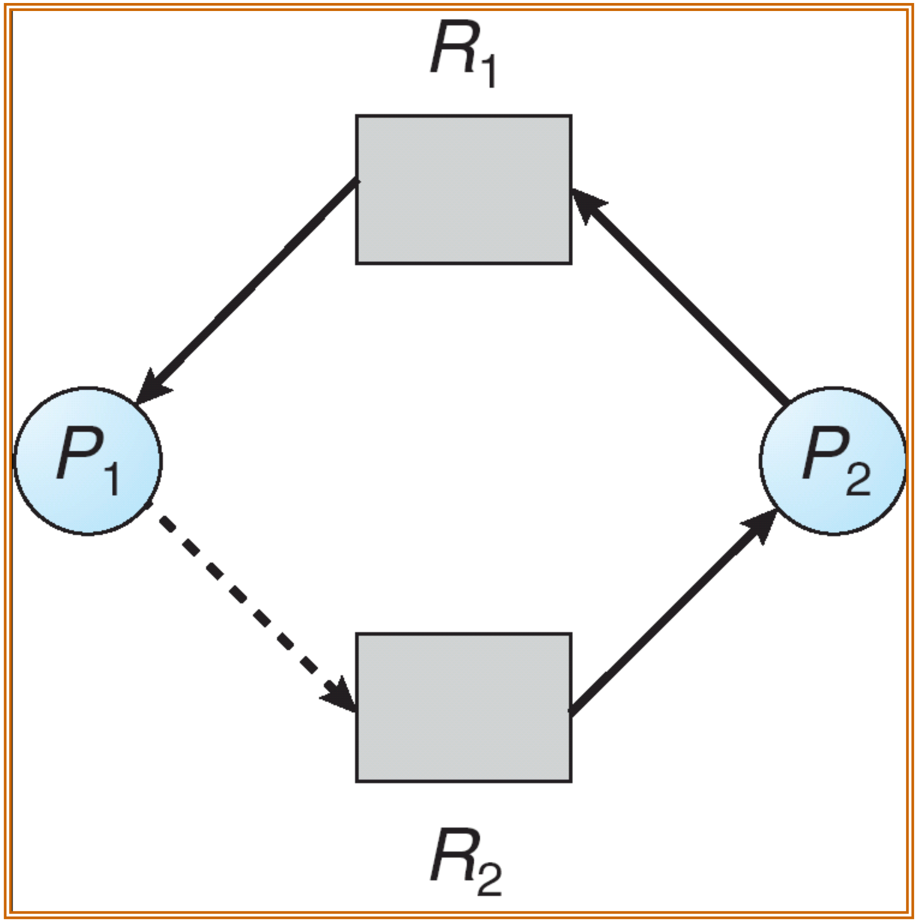
- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_i ***MAY*** request resource R_j
  - represented by a dashed line
- Claim edge converted to request edge when a process requests a resource
- When a resource is released by a process, the assignment edge is reconverted to a claim edge
- Resources must be claimed *a priori* in the system
- A request is granted **if it does not cause a cycle**
  - Safe state

Resource-Allocation Graph for Deadlock Avoidance



Safe

Unsafe State in Resource-Allocation Graph



P2 request R2

➔Granting the request will cause a cycle
➔ the request cannot be granted

*This method does not support resources with multiple instances

Banker's Algorithm

- Supports resources with multiple instances
- Each process must *a priori* claim maximum use
- When a process requests a resource it may have to wait

Data Structures for the Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

- *Available*: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- *Max*: $n \times m$ matrix. If $Max\,[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- *Allocation*: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
- *Need*: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.
  - $Need\,[i,j] = Max[i,j] – Allocation\,[i,j]$

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process i

$Request_i[j] = k$ ➔ process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request_i$

   $Allocation_i = Allocation_i + Request_i$

   $Need_i = Need_i – Request_i$
4. Invoke the safety algorithm
   - *If safe $\Rightarrow$ the resources are allocated to Pi*
   - *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively

   Initialization--

   Work = Available

   Finish [*i*] = *false* for *i* = 0, 1, 2, …, *n*

2. Find an *i* such that both

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$ // *can allow it to finish*

   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$ // *Pi releases its resources*

   Finish[*i*] = *true*

   go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types
  - *A* (10 instances), *B* (5 instances), and *C* (7 instances).
- Snapshot at time T_0

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix. Need is defined to be Max - Allocation

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Example P_1 Request (1,0,2) (Cont.)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2))⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement
- Then,
  - can request for (3,3,0) by P4 be granted? No, > available
  - can request for (0,2,0) by P0 be granted? No, unsafe

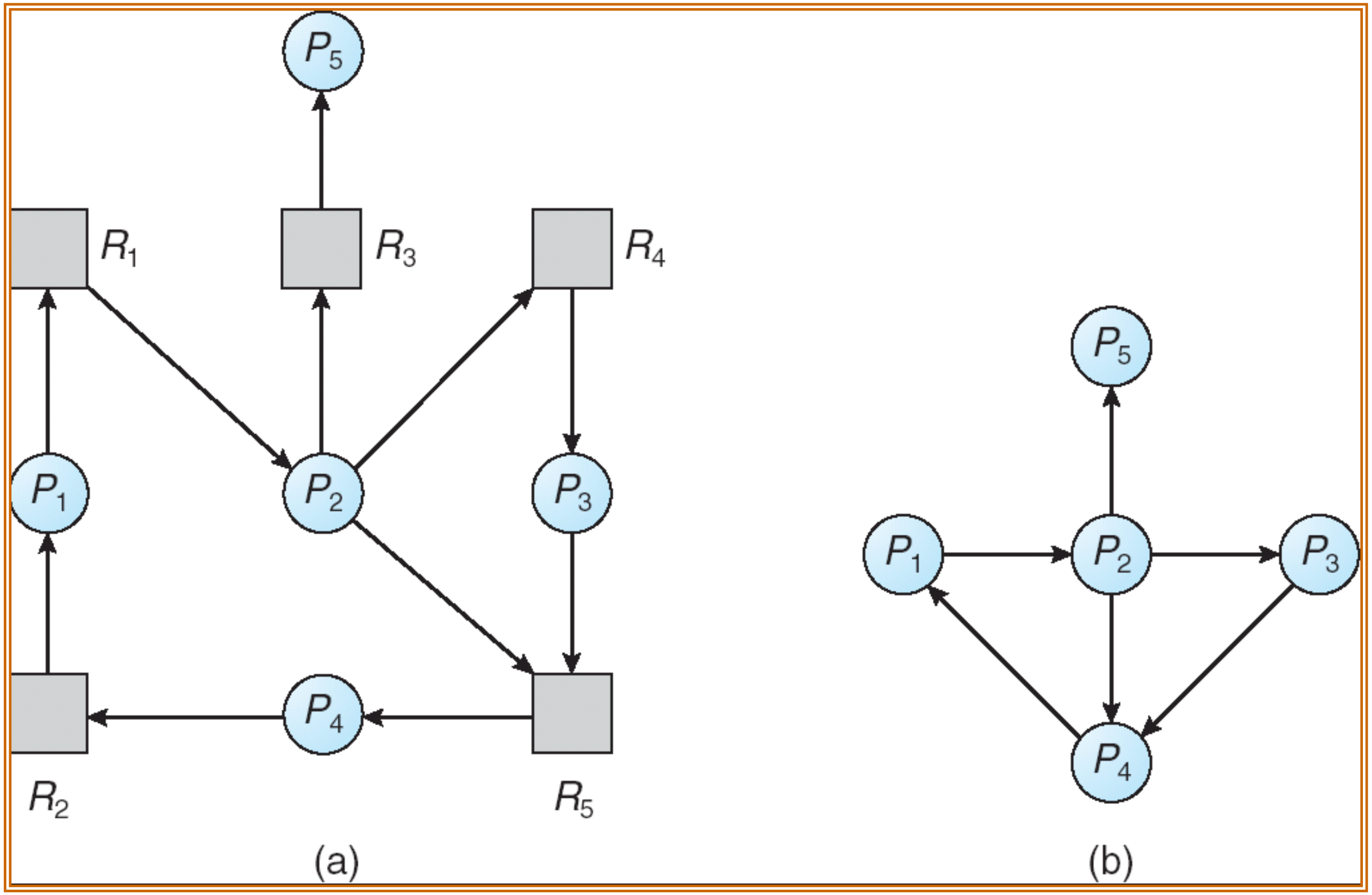
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance for Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph **Corresponding wait-for graph**

Several Instances of a Resource Type

- *Available*: A vector of length *m* indicates the number of available resources of each type
- *Allocation*: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process
- *Request*: An *n* x *m* matrix indicates the **current request** of each process. If *Request* [*i*, *j*] = *k*, then process P_i is requesting *k* more instances of resource type R_j

- Different from the banker's algorithm used for deadlock avoidance
  - Check **current request**, instead of **future need**
  - **Not always** check when a request is made

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively.

   Initial condition:

   (a) $Work = Available$

   (b) For $i = 1,2, \ldots, n$,

   If $Allocation_i \neq 0$, $Finish[i] = \text{false}$

   Otherwise, $Finish[i] = true$

2. Find an index i such that both:

   (a) $Finish[i] == false$

   (b) $Request_i \leq Work$

If no such i exists, go to step 4.

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i]$ == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i]$ == *false*, then P_i is in a deadlock state.

Optimistically assumes that Pi will release its resources after the request.
If the assumption is not correct, deadlock may occur later.
- detected by the later invocation of the detection algorithm

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in a deadlock state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type C.

	Request
	$A\ B\ C$
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- Is the system in a safe state?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke? It depends on
  - How often a deadlock is likely to occur?
  - How many processes will be affected?

- Possible ways
  - Invoked at each request that needs to wait
  - Invoked periodically
  - Invoked at specific situations
    - E.g., when the system throughput suddenly drops

Recovery from Deadlock

- Two general approaches
  - Process Termination
  - Resource Preemption

Recovery from Deadlock: Process Termination

- Abort all deadlock processes
- Abort one process at a time until the deadlock cycle is eliminated
- Selecting a victim
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process holds
  - Resources a process needs to complete

Recovery from Deadlock: Resource Preemption

- Preempt some resources from some processes and give the resources to other processes
- Selecting a victim – minimize cost, similar to the previous slide
- Rollback – return to some *safe* state, restart process from that state
- Starvation – a process may always be selected as the victim
  - May need to include number of rollback in the cost factor